Universidad Nacional de Córdoba
Facultad de Ciencias Exactas, Físicas y Naturales
Escuela de Ingeniería en Computación

Trabajo Práctico 1 (versión 2)

Programación Concurrente

Integrantes: Cabrera, Augusto Gabriel
Mansilla, Josías Leonel
Moroz, Esteban Mauricio
Pallardó, Agustín
Villar, Federico Ignacio
Profesores: Ludemann, Mauricio
Ventre, Luis Orlando

Fecha de entrega: 25 de abril
Córdoba

Resumen

En el siguiente informe se adjunta todo el procedimiento de trabajo implementado para la realización del primer trabajo práctico de la materia Programación Concurrente.

En un inicio, se plantea el diagrama de clases del programa a implementar, para luego, esbozar la estructura básica del mismo implementando lo solicitado por el enunciado. Una vez obtenido un programa funcional, se procedió a realizar diferentes pruebas para corroborar el correcto funcionamiento del mismo y encontrar posibles falencias, entre las que podrían haber problemas de concurrencia.

Finalmente, se analizaron los resultados obtenidos y se elaboraron conclusiones en base a los mismos. En este informe se encuentran además, diferentes explicaciones que contrinuyen al correcto entendimiento del programa desarrollado, así como a la comprobación del correcto funcionamiento del mismo.

Índice de Contenidos

Índice de Figuras

Índice de Códigos

1. Consignas

1.1. Enunciado

En un sistema de gestión de imágenes, existe una funcionalidad que se encarga de mejorar la calidad de los archivos obtenidos, previo a la entrega final al cliente. Esta funcionalidad ha quedad obsoleta, por lo tanto se solicita realizarlo nuevamente, teniendo en cuenta los siguientes lineamientos de diseño.

Dicha funcionalidad se divide en **cuatro procesos**.

El **primer proceso** se encarga de cargar las imágenes en un contenedor, en su estado original. Existen **dos hilos** que ejecutan este proceso, demorando un tiempo aleatorio en ms para obtener la imagen y depositarla en el contenedor. Ambos hilos dejan las imágenes en el mismo contenedor.

El **segundo proceso** tiene por objetivo mejorar la iluminación de cada imagen. En este caso, **tres hilos** ejecutan este proceso, demorando un **tiempo aleatorio** en ms para mejorar la iluminación de cada imagen. Cada hilo debe tomar una **imagen aleatoria** del contenedor por vez, y no puede tomar más de una vez la misma imagen. En el tiempo que demora en mejorar la imagen no debe bloquear otros hilos que quieran tomar otras imágenes para mejorar. Cada imagen debe ser mejorada por **todos los hilos** (los 3 del presente proceso) para que el siguiente proceso pueda tomarla.

El **tercer proceso** debe ajustar las imágenes al tamaño final solicitado. Este proceso tiene que ejecutarse después que la imagen ya fue mejorada. **Tres hilos** son los encargados de ejecutar este proceso, tomando cada uno una **imagen aleatoria** del contenedor por vez para ajustarla, y demorando un **tiempo aleatorio** en ms. En el tiempo que demora en ajustar la imagen no debe bloquear otros hilos que quieran tomar otras imágenes para ajustar. Una imagen solo puede ser ajustada una sola vez.

El **cuarto proceso** toma las imágenes ya mejoradas y ajustadas, les hace una copia en el contenedor final, y luego las elimina del contenedor final, y luego las elimina del contenedor inicial. **Dos hilos** ejecutan este proceso, y demoran un tiempo en ms en realizar su trabajo. Cada hilo toma de a una imagen por vez de **manera aleatoria**.

1.1.1. Consideraciones

- Los objetos tienen un atributo ïmprovements", el cual registra cuántos hilos han tomado el archivo para mejorarlo (proceso 2).
- Cuando se menciona un tiempo aleatorio en ms, el mismo debe ser no nulo y a elección del grupo.
- Una imagen no puede pasar a ser ajustada sin antes haber sido mejorada por los 3 hilos del segundo proceso.
- Al finalizar la ejecución es necesario verificar cuántas imágenes movió del contenedor inicial hacia el contenedor final, cada hilo del cuarto proceso.
- El sistema debe contar con un LOG con fines estadísticos, el cual registre cada 500 ms en un archivo:
  - Cantidad de imágenes insertadas en el contenedor.
  - Cantidad de imágenes mejoradas complementamente.

- Cantidad imágenes ajustadas.
- Cantidad de imágenes que han finalizado el último proceso.
- El estado de cada hilo del sistema.

1.1.2. Ejercicios

1. Hacer un diagrama de clases que modele el sistema de datos con TODOS los actores y partes.
2. Hacer un solo diagrama de secuencias que muestre la siguiente interacción:
   A Con el contenedor inicial vacío, la inserción de un archivo.
   B La interacción de un hilo del segundo proceso al intentar acceder al archivo para la mejora.
   C La interacción de un hilo del tercer proceso cuando accede a un archivo para ajustar el tamaño de la imagen.
3. Las demoras del sistema en sus cuatro procesos deben configurarse de tal manera de poder procesar 100 imágenes (desde la inserción en el primer contender hasta que son movidas al contenedor final) en un periodo mínimo de 10 segundos y máximo de 20 segundos.
4. Hacer un análisis detallado de los tiempos que el programa demora. Luego contrastarlo con múltiples ejecuciones, obteniendo las conclusiones pertinentes.
5. El grupo debe poder explicar los motivos de los resultados obtenidos. Y los tiempos del sistema.
6. Debe haber una clase Main que al correrla, inicie los hilos.

2. Código de Java

2.1. Funcionamiento

Se crearon las instancias de las clases y variables que se utilizarán. Luego, se inicializan los arreglos de las distintas clases vivas. Se lo hizo de la siguiente forma: 2 hilos **Loader**, 3 hilos **Improver**, 3 hilos **Resizer** y 2 hilos **Cloner**. Además se crea un objeto de la clase **Log**. Posteriormente, se inicia el funcionamiento de todos los hilos y comienza la ejecución del programa.

2.1.1. Clases vivas

- **Loader**: se encarga de crear instancias de la clase **Image** y guardarlas dentro de **initContainer**. Esta acción se repite de manera iterada mientras que la cantidad total de datos agregados al **initContainer** sea menor a la establecida previamente como objetivo (**targetAmountOfData**). El **Loader** posee un contador que lleva el registro de la cantidad total de imágenes que ha agregado al **initContainer** y se puede así consultar su valor a través del método **getLoadedimages()**. Si el **initContainer** está lleno y además se le intentan cargar más imágenes, se lanza una excepción para evitar que otro hilo entre al bloque **synchronized** del método **Load()** y no agregue otra imagen al contenedor inicial. Finalmente, el hilo duerme 50 milisegundos entre iteraciones.

- **Improver**: se encarga de chequear por única vez cada instancia de la clase **Imagen** que está almacenada en el **initContainer**. Esto se va a repetir mientras que el proceso de creación de datos no haya sido finalizado y el **initContainer** contenga elementos. Los hilos **Improvers** chequean que la imagen tomada con el método **isImproved()**. Si esta última no fue modificada por el **Improver** correspondiente, lo agrega a la lista enlazada **Improvements** para que no se vuelva a modificar (ser tomada) por el mismo hilo dos veces. Una vez que los tres hilos mejoren la imagen, el proceso **improveByThread()** cambia el valor del campo **iAmImproved** a **true**, indicando que la imagen en cuestión ya fue mejorada tres veces, como se requería, garantizando que fue mejorada una vez por cada uno de los hilos. Finalmente, el hilo duerme por 100 milisegundos entre iteraciones.

- **Resizer**: se encarga de tomar una imagen aleatoria del contenedor inicial y corroborar que fue mejorada por los tres **Improver** haciendo uso del método getter llamado **getAmIImproved()**. Luego, una vez encontrada una imagen que cumple con el requisito, se procede a realizarle el **resize()**, que cambia el valor del campo **resized** a **true**, indicando así que la imagen ya fue reescalada y aumenta el valor del contador **totalImagesResized** con el método **increaseImageResizer()**. Por último, realiza un sleep de 100 milisegundos.

- **Cloner**: se encarga de tomar una imagen aleatoria del contenedor inicial, corroborar que fue reescalada, y, si lo fue, mediante el proceso **copyAndDelete()** la toma y la copia en el contenedor final, borrándola del contenedor inicial. Para ello, se hace uso del método **tryCloneToFinalContainer()** para evitar problemas de concurrencia y comprobar si ya fue agregada al contenedor final. Esta comprobación se hace porque si dos hilos ingresan al método **clone()** a la vez, uno de los dos cambia el valor de la variable **increaseImageClone()**, esto es para poder mostrar por **Log** la cantidad de imágenes clonadas por cada hilo. Finalmente se duerme por 50 milisegundos.

- **Log**: se encarga de imprimir en un archivo de texto llamado **Estadistica.txt** la información acerca del estado del programa en intervalos de tiempo de 500 milisegundos.

2.1.2. Implementación

El programa desarrollado cuenta con las siguientes clases:

- **Cloner**
- **Container**
- **FinalContainer**
- **Image**
- **Improver**
- **InitConainer**
- **Loader**
- **Log**
- **Main**
- **Resizer**

Se implementan de la siguiente forma:

Código 1: Clase Main

```
1  package TP1;
2
3  // Clase principal
4  public class Main {
5    // Se indica la cantidad de hilos para cada proceso
6    private static final int numberOfLoaders = 2;
7    private static final int numberOfImprovers = 3;
8    private static final int numberOfResizers = 3;
9    private static final int numberOfClones = 2;
10   private static final int targetAmountOfData = 100;
11
12   public static void main(String[] args) {
13     // Se inicia el contador de tiempo
14     long start = System.currentTimeMillis();
15     // Se limpia el archivo del Log
16     Log.clearFile();
17     // Se instancian los contenedores
18     InitContainer initContainer = new InitContainer(targetAmountOfData);
19     FinalContainer finalContainer = new FinalContainer(targetAmountOfData);
20     // Se crean arreglos con las diferentes instancias
21     Loader[] loaders = new Loader[numberOfLoaders];
22     Cloner[] cloners = new Cloner[numberOfClones];
23     Resizer[] resizers = new Resizer[numberOfResizers];
```

```
24          Improver[] improvers = new Improver[numberOfImprovers];
25          // Se crean arreglos de hilos
26          Thread[] loadersThreads = new Thread[numberOfLoaders];
27          Thread[] improversThreads = new Thread[numberOfImprovers];
28          Thread[] resizerThreads = new Thread[numberOfResizers];
29          Thread[] clonersThreads = new Thread[numberOfClones];
30          // Se inicia el Log
31          TP1.Log log = new TP1.Log(targetAmountOfData, initContainer, finalContainer, loaders,
    ↪ improvers, resizers,
32                  cloners, loadersThreads, improversThreads, resizerThreads, clonersThreads);
33          // Se crean los hilos
34          for (int i = 0; i < numberOfLoaders; i++) {
35            loaders[i] = new Loader(initContainer, "TP1.Loader " + i);
36            loadersThreads[i] = new Thread(loaders[i]);
37            loadersThreads[i].setName(loaders[i].getName() + " (Thread ID: " +
    ↪ loadersThreads[i].getId() + ")");
38          }
39          for (int i = 0; i < numberOfImprovers; i++) {
40            improvers[i] = new Improver(initContainer, "TP1.Improver " + i, numberOfImprovers,
    ↪ targetAmountOfData);
41            improversThreads[i] = new Thread(improvers[i]);
42            improversThreads[i].setName(improvers[i].getName() + " (Thread ID: " +
    ↪ improversThreads[i].getId() + ")");
43          }
44          for (int i = 0; i < numberOfResizers; i++) {
45            resizers[i] = new Resizer(initContainer, "TP1.Resizer " + i , targetAmountOfData);
46            resizerThreads[i] = new Thread(resizers[i]);
47            resizerThreads[i].setName(resizers[i].getName() + " (Thread ID: " +
    ↪ resizerThreads[i].getId() + ")");
48          }
49          for (int i = 0; i < numberOfClones; i++) {
50            cloners[i] = new Cloner(initContainer, finalContainer, "TP1.Cloner " + i);
51            clonersThreads[i] = new Thread(cloners[i]);
52            clonersThreads[i].setName(cloners[i].getName() + " (Thread ID: " +
    ↪ clonersThreads[i].getId() + ")");
53          }
54          // Se les hace el start() a cada hilo
55          for (Thread loadersThread : loadersThreads) {
56            loadersThread.start();
57          }
58          for (Thread improverThread : improversThreads) {
59            improverThread.start();
60          }
61          for (Thread resizersThread : resizerThreads) {
62            resizersThread.start();
63          }
64          for (Thread clonerThread : clonersThreads) {
65            clonerThread.start();
66          }
67          // El Log empieza a tomar datos
68          log.start();
```

```
69          // Se le hace join() a los hilos para esperar a que mueran y continuar con el main
70          try {
71            for (Thread waiting : loadersThreads) {
72              waiting.join();
73            }
74            for (Thread waiting : improversThreads) {
75              waiting.join();
76            }
77            for (Thread waiting : resizerThreads) {
78              waiting.join();
79            }
80            for (Thread waiting : clonersThreads) {
81              waiting.join();
82            }
83          } catch (InterruptedException e) {
84            e.printStackTrace();
85          }
86          // Se le pide la interrupción al Log
87          log.interrupt();
88          // Se finaliza la cuenta del cronómetro iniciado
89          long finish = System.currentTimeMillis();
90          long timeElapsed = finish - start;
91          // Se imprime información de la ejecución
92          // La información más completa aparece en el Log estadístico
93          System.out.println("\nExecution Complete!");
94          System.out.println("\nThread's states:");
95          for (Thread loaderThread : loadersThreads) {
96            System.out.println(loaderThread.getName() + ": " + loaderThread.getState().name());
97          }
98          for (Thread improverThread : improversThreads) {
99            System.out.println(improverThread.getName() + ": " +
    ↪ improverThread.getState().name());
100         }
101         for (Thread resizerThread : resizerThreads) {
102           System.out.println(resizerThread.getName() + ": " + resizerThread.getState().name());
103         }
104         for (Thread clonerThread : clonersThreads) {
105           System.out.println(clonerThread.getName() + ": " + clonerThread.getState().name());
106         }
107         int totalLoadedImages = 0;
108         int totalImprovements = 0;
109         int totalResizing = 0;
110         int totalClonedImages = 0;
111         for (Loader load : loaders) {
112           totalLoadedImages += load.getLoadedImages();
113         }
114         for (Improver improver : improvers) {
115           totalImprovements += improver.getTotalImagesImprovedByThread();
116         }
117         for (Resizer resizer : resizers) {
118           totalResizing += resizer.getTotalImagesResized();
```

```
119         }
120         for (Cloner cloner : cloners) {
121           totalClonedImages += cloner.getClonedImages();
122         }
123         System.out.println("\nPerformance stats:");
124         System.out.println("Elapsed Time: " + (float) (timeElapsed / 1000.00) + " seconds");
125         System.out.println("Final InitContainer size: " + initContainer.getSize());
126         System.out.println("Final FinalContainer size: " + finalContainer.getSize());
127         System.out.println("Total Loaded Images: " + totalLoadedImages);
128         System.out.println("Total improvements: " + totalImprovements);
129         System.out.println("Total resizing: " + totalResizing);
130         System.out.println("Total cloned Images " + totalClonedImages);
131       }
132     }
133
```

Código 2: Clase Cloner

```
1   package TP1;
2
3   import java.util.concurrent.TimeUnit;
4
5   // Clase que clona las imágenes y las carga al contenedor final
6   public class Cloner implements Runnable {
7     private final InitContainer initContainer; // Contenedor inicial
8     private final FinalContainer finalContainer; // Contenedor final
9     private final String name; // Nombre del cloner
10    private Image lastClonedImage; // Última imagen clonada
11    private int clonedImages; // Cantidad de imágenes clonadas
12
13    // Constructor
14    public Cloner(InitContainer initContainer, FinalContainer finalContainer, String name) {
15      this.initContainer = initContainer;
16      this.finalContainer = finalContainer;
17      this.name = name;
18      lastClonedImage = null;
19      clonedImages = 0;
20    }
21
22    // Sobreescritura del método run()
23    @Override
24    public void run() {
25      while (initContainer.getSize() > 0 || initContainer.isLoadNotCompleted()) {
26        try {
27          lastClonedImage = initContainer.getImage();
28          if (lastClonedImage != null) {
29            if (lastClonedImage.isResized()) {
30              if (!lastClonedImage.amIDeletedFromInitContainer()) {
31                if (lastClonedImage.tryCloneToFinalContainer()) {
32                  if (finalContainer.Clone(initContainer.copyAndDelete(lastClonedImage),
    ↪ this)) {
```

```
                        lastClonedImage.setClonedToFinalContainer(true);
                        increaseImageClone();
                        TimeUnit.MILLISECONDS.sleep(50);
                      }
                    }
                  }
                }
              }
            } catch (Exception e) {
              e.printStackTrace();
            }
          }
        }

        // Getter del campo name
        public String getName() {
          return name;
        }

        // Getter de la cantidad de imágenes clonadas
        public int getClonedImages() {
          return clonedImages;
        }

        // Método que incrementa la cantidad de imágenes clonadas
        public void increaseImageClone() {
          clonedImages++;
        }
    }

```

Código 3: Clase Container

```
package TP1;

import java.util.LinkedList;

// Clase abstracta contenedor
public abstract class Container {

    protected LinkedList<Image> container;
    protected int targetAmountOfImages;

    public Container() {
      container = new LinkedList<>();
    }

    public int getSize() {
      return container.size();
    }
}
```

```
19
```

Código 4: Clase FinalContainer

```
1   package TP1;
2
3   // Clase contenedor final
4   public class FinalContainer extends Container {
5
6     private boolean cloneCompleted; // Boolean que representa la completión de la carga
7     private final int targetAmountOfImages; // Cantidad objetivo de imágenes para guardar en el
    ↪ contenedor
8     private int amountOfImages; // Cantidad de imágenes
9
10    // Constructor
11    public FinalContainer(int targetAmountOfImages) {
12      this.amountOfImages = 0;
13      this.targetAmountOfImages = targetAmountOfImages;
14      cloneCompleted = false;
15    }
16
17    // Método que clona las imágenes del contenedor inicial al final
18    public synchronized boolean Clone(Image image, Cloner cloner) throws
    ↪ FullContainerException {
19      try {
20        if (!cloneCompleted && image != null && image.isResized()) {
21          this.container.addLast(image);
22          amountOfImages++;
23          System.out.printf("[FinalContainer (Size: %d)] %s Image cloned <ID: %d \n",
24              this.container.size(),
25              Thread.currentThread().getName(), image.getId());
26          if (amountOfImages == targetAmountOfImages) {
27            cloneCompleted = true;
28            cloner.increaseImageClone();
29          }
30          else if (amountOfImages > targetAmountOfImages) {
31            throw new FullContainerException("Contenedor Excedido");
32          }
33        }
34      } catch (Exception e) {
35        e.printStackTrace();
36      }
37      return !cloneCompleted;
38    }
39  }
40
```

Código 5: Clase Image

```
1   package TP1;
```

```
2
3   import java.util.ArrayList;
4   import java.util.List;
5
6   // Clase que representa las imágenes a procesar
7   public class Image {
8
9     private final List<Improver> improvements; // Lista de los hilos improver que mejoraron la
    ↪ imagen
10    private final int id; // Identificador de la imagen
11    private static int generator = 0; // Generador para el nombre
12    private static final Object key = new Object(); // Llave
13    private boolean resized; // Boolean que indica si la imagen se reescaló
14    private boolean clonedToFinalContainer; // Booleano que indica si la imagen fue clonada al
    ↪ contenedor final
15
16    private final Object keyImprove = new Object(); // Llave para el sincronismo del improve
17    private final Object keyResize = new Object(); // Llave para el sincronismo de resize
18    private final Object keyClone = new Object(); // Llave para el sincronismo del clonado
19
20    private boolean iAmDeletedFromInitContainer; // Booleano para indicar si la imagen fue
    ↪ eliminada del contenedor inicial
21
22    private boolean iAmImproved; // Booleano para verificar si la imagen está mejorada
23
24    // Se genera un nuevo id, se usa el generador de la clase (static)
25    private static int newId() {
26      synchronized (key) {
27        return generator++;
28      }
29    }
30
31    public Image() {
32      improvements = new ArrayList<>();
33      resized = false;
34      id = newId();
35      clonedToFinalContainer = false;
36      iAmImproved = false;
37      iAmDeletedFromInitContainer = false;
38    }
39
40    // Método Getter de la lista de hilos que mejoraron la imagen
41    public List<Improver> getImprovements() {
42      return improvements;
43    }
44
45    // Setter del campo resized
46    public void setResized(boolean resized) {
47      this.resized = resized;
48    }
49
```

```
50      // Setter del campo clonedToFinalContainer
51      public void setClonedToFinalContainer(boolean clonedToFinalContainer) {
52        this.clonedToFinalContainer = clonedToFinalContainer;
53      }
54
55      // Constructor
56      public Image(List<Improver> improvements, boolean resized, int id, boolean
    ↪ clonedToFinalContainer,
57          boolean iamImproved) {
58        this.improvements = improvements;
59        this.resized = resized;
60        this.id = id;
61        this.clonedToFinalContainer = clonedToFinalContainer;
62        this.iAmImproved = iamImproved;
63      }
64
65      // Getter del campo iAmDeletedFromInitContainer
66      public boolean amIDeletedFromInitContainer() {
67        return iAmDeletedFromInitContainer;
68      }
69
70      // Método para mejorar la imagen por un hilo Improver
71      public void improveByThread(Improver improver) {
72        synchronized (keyImprove) {
73          improvements.add(improver);
74          if (improvements.size() == improver.getTotalThreadsImprovements()) {
75            this.setIamImprove();
76          }
77        }
78      }
79
80      // Getter que indica si la imagen fue mejorada por un cierto hilo Improver que recibe como
    ↪ parámetro
81      public boolean isImprovedByThread(Improver improver) {
82        return improvements.contains(improver);
83      }
84
85      // Método que reescala la imagen
86      public boolean resize() {
87        synchronized (keyResize) {
88          if (!isResized()) {
89            setResized(true);
90            return true;
91          } else {
92            return false;
93          }
94        }
95      }
96
97      // Getter del id de la imagen
98      public int getId() {
```

```
99          return id;
100       }
101
102       // Setter del improve
103       public void setIamImprove() {
104         iAmImproved = true;
105       }
106
107       // Getter que indica si la imagen fue reescalada
108       public boolean isResized() {
109         return resized;
110       }
111
112       // Método que intenta clonar la imagen al contenedor inicial
113       public boolean tryCloneToFinalContainer() {
114         synchronized (keyClone) {
115           if (!iAmDeletedFromInitContainer) {
116             iAmDeletedFromInitContainer = true;
117             return true;
118           } else {
119             return false;
120           }
121         }
122       }
123
124       // Getter que indica si la imagen fue mejorada
125       public boolean getAmIImproved() {
126         return iAmImproved;
127       }
128
129       // Getter que indica si la imagen fue reescalada
130       public boolean getAmIResized() {
131         return resized;
132       }
133     }
134
```

Código 6: Clase Improver

```
1   package TP1;
2
3   import java.util.concurrent.TimeUnit;
4
5   // Clase que se encarga de mejorar las imágenes
6   public class Improver implements Runnable {
7
8     private final InitContainer initContainer; // Contenedor inicial
9
10    private static boolean finishImprove = false; // Booleano para indicar si se completó la mejora
11
12    private int totalImagesImprovedByThread; // Cantidad de imágenes mejoradas por el hilo
```

```
13
14      private final String name; // Nombre
15
16      private static int totalImprovedImages = 0; // Imágenes mejoradas por esta clase
17
18      private final int totalThreadsImprovements;
19
20      private Image lastImprovedImage; // Última imagen mejorada
21
22      private final int targetAmountOfData; // Cantidad de imágenes objetivo
23
24      public Improver(InitContainer initContainer, String name, int totalThreadsImprovements, int
    ↪ targetAmountOfData) {
25        this.initContainer = initContainer;
26        this.name = name;
27        this.totalThreadsImprovements = totalThreadsImprovements;
28        totalImagesImprovedByThread = 0;
29        lastImprovedImage = null;
30        this.targetAmountOfData=targetAmountOfData;
31
32      }
33
34      // Sobreescritura del método run()
35      @Override
36      public void run() {
37        while (!finishImprove) {
38          try {
39            lastImprovedImage = initContainer.getImage();
40            if (lastImprovedImage != null) {
41              if (!lastImprovedImage.isImprovedByThread(this)) {
42                System.out.println("Improved image: " + lastImprovedImage.getId() + ", thread: "
43                    + Thread.currentThread().getName());
44                lastImprovedImage.improveByThread(this);
45                TimeUnit.MILLISECONDS.sleep(100);
46                totalImagesImprovedByThread++;
47                increaseTotalImprovedImages();
48                if(getTotalImprovedImages() ==
    ↪ targetAmountOfData*totalThreadsImprovements){
49                  finishImprove=true;
50                }
51              }
52            }
53          } catch (Exception e) {
54            e.printStackTrace();
55          }
56        }
57      }
58
59      // Getter del name()
60      public String getName() {
61        return name;
```

```
62      }
63
64      // Getter de la cantidad de imágenes mejoradas por el proceso (clase)
65      public static int getTotalImprovedImages() {
66        return totalImprovedImages;
67      }
68
69      // Método que incrementa la cantidad de imágenes mejoradas por el proceso (clase)
70      public synchronized static void increaseTotalImprovedImages(){
71        totalImprovedImages++;
72      }
73
74      // Getter de la cantidad de mejoras hechas
75      public int getTotalThreadsImprovements() {
76        return totalThreadsImprovements;
77      }
78
79      // Getter de la cantidad de imágenes mejoradas por el hilo
80      public int getTotalImagesImprovedByThread() {
81        return totalImagesImprovedByThread;
82      }
83    }
84
```

Código 7: Clase InitContainer

```
1   package TP1;
2
3   import java.util.Random;
4
5   // Contenedor inicial del proceso
6   public class InitContainer extends Container {
7
8     private boolean loadCompleted; // Boolean que indica si la carga de imágenes está completa
9     private final int targetAmountOfImages; // Cantidad objetivo de imágenes
10    private int amountOfImages; // Cantidad actual de imágenes
11
12    private final Object keyLoad = new Object(); // Llave para sincronismo en la carga
13
14    public InitContainer(int targetAmountOfImages) {
15      loadCompleted = false;
16      this.targetAmountOfImages = targetAmountOfImages;
17      this.amountOfImages = 0;
18    }
19
20    // Método de carga de imágenes
21    public boolean load(Image image, Loader loader, int amount) throws FullContainerException {
22      synchronized (keyLoad) {
23        if (!loadCompleted) {
24          container.addLast(image);
25          amountOfImages++;
```

```
26              System.out.println("Loaded image: " + image.getId());
27              if (amountOfImages == targetAmountOfImages) {
28                loadCompleted = true;
29                loader.setLoadedImages(amount + 1);
30              }
31              else if (amountOfImages > targetAmountOfImages) {
32                throw new FullContainerException("Contenedor Excedido");
33              }
34            }
35            return loadCompleted;
36          }
37        }
38
39        // Método que retorna una imagen aleatoria del contenedor
40        public synchronized Image getImage() {
41          if (container.size() > 0) {
42            int aux = new Random().nextInt(container.size());
43            return container.get(aux);
44          } else {
45            return null;
46          }
47        }
48
49        // Getter que permite ver si la carga no está completa
50        public boolean isLoadNotCompleted() {
51          return !loadCompleted;
52        }
53
54        // Método para copiar y eliminar imagen de este contenedor
55        public synchronized Image copyAndDelete(Image image) {
56          if (container.size() > 0 && image.amIDeletedFromInitContainer()) {
57            Image forClone = new Image(image.getImprovements(), image.getAmIResized(),
    ↪ image.getId(), true,
58                image.getAmIImproved());
59            this.container.remove(image);
60            System.out.printf("Imagen copiada y borrada del contenedor inicial: " + forClone.getId()
    ↪ + "\n");
61            return forClone;
62          } else {
63            return null;
64          }
65        }
66
67        // Getter de la cantidad de imágenes en el contenedor
68        public int getAmountOfImages() {
69          return amountOfImages;
70        }
71      }
72
```

Código 8: Clase Loader

```java
package TP1;

import java.util.concurrent.TimeUnit;

// Clase que se encarga de cargar las imágenes en el contenedor inicial
public class Loader implements Runnable {

  private int loadedImages; // Cantidad de imágenes cargadas

  private final InitContainer initContainer; // Contenedor inicial

  private final String name; // Nombre

  public Loader(InitContainer initContainer, String name) {
    this.initContainer = initContainer;
    this.name = name;
    loadedImages = 0;
  }

  // Sobreescritura del método run()
  @Override
  public void run() {
    while (initContainer.isLoadNotCompleted()) {
      try {
        if (!initContainer.load(new Image(), this, loadedImages)) {
          increaseImageLoad();
          TimeUnit.MILLISECONDS.sleep(50);
        }
      }
      // Se deja este catch por si por alguna razón aparece alguna excepción, de todos modos no aparecen
      catch (Exception e) {
        e.printStackTrace();
      }
    }
  }

  // Setter de cantidad de imágenes cargadas
  public void setLoadedImages(int loadedImages) {
    this.loadedImages = loadedImages;
  }

  // Getter del name del Loader
  public String getName() {
    return name;
  }

  // Getter de la cantidad de imágenes cargadas
  public int getLoadedImages() {
    return loadedImages;
```

```
50        }
51
52        // Método para incrementar la cantidad de imágenes cargadas
53        public void increaseImageLoad() {
54          loadedImages++;
55        }
56
57      }
58
```

Código 9: Clase Log

```
1     package TP1;
2
3     import java.io.FileWriter;
4     import java.io.IOException;
5     import java.io.PrintWriter;
6     import java.util.*;
7     import java.util.concurrent.TimeUnit;
8
9     // Clase que crea el Log estadístico para la ejecución del programa
10    public class Log extends Thread {
11      private final int targetAmountOfData; // Objetivo de imágenes
12      private final Date initTime; // Fecha inicial
13      private final InitContainer initContainer; // Contenedor inicial
14      private final FinalContainer finalContainer; // Contenedor final
15
16      // Hilos en arreglos
17      private final Loader[] loaders;
18      private final Improver[] improvers;
19      private final Resizer[] resizers;
20
21      private final Cloner[] cloners;
22
23      private final Thread[] loadersThreads;
24      private final Thread[] improversThreads;
25      private final Thread[] resizersThreads;
26
27      private final Thread[] clonersThreads;
28
29      // Método que crea un archivo txt limpio
30      public static void clearFile() {
31        try {
32          PrintWriter pw_log = new PrintWriter(".//Estadistica.txt");
33          pw_log.print("");
34          pw_log.close();
35        } catch (IOException e) {
36          e.printStackTrace();
37        }
38      }
39
```

```
40        // Constructor
41        public Log(int targetAmountOfData, InitContainer initContainer,
42            FinalContainer finalContainer,
43            Loader[] loaders,
44            Improver[] improvers,
45            Resizer[] resizers,
46            Cloner[] cloners,
47            Thread[] loadersThreads,
48            Thread[] improversThreads,
49            Thread[] resizersThreads,
50            Thread[] clonersThreads) {
51          this.initContainer = initContainer;
52          this.finalContainer = finalContainer;
53          this.improvers = improvers;
54          this.loaders = loaders;
55          this.resizers = resizers;
56          this.cloners = cloners;
57          this.improversThreads = improversThreads;
58          this.resizersThreads = resizersThreads;
59          this.clonersThreads = clonersThreads;
60          this.loadersThreads = loadersThreads;
61          this.targetAmountOfData = targetAmountOfData;
62          initTime = new Date();
63        }
64
65        // Sobreescritura del método run()
66        @Override
67        public void run() {
68          while (finalContainer.getSize() <= targetAmountOfData) {
69            try {
70              writeLog();
71              TimeUnit.MILLISECONDS.sleep(500);
72            } catch (InterruptedException e) {
73              writeLog();//?
74              break;
75            }
76          }
77        }
78
79        // Se escribe el Log
80        private void writeLog() {
81          try {
82            PrintWriter pw_log = new PrintWriter(new FileWriter(".//Estadistica.txt", true));
83            pw_log.print("*---------------------------------------------------------------------------*\n");
84            pw_log.printf("Execution time: %.3f [Seg]\n", (float) (new Date().getTime() -
      ↪ initTime.getTime()) / 1000);
85            pw_log.printf("InitContainer size at this moment: %d\n", initContainer.getSize());
86            pw_log.printf("InitContainer size: %d\n", initContainer.getAmountOfImages());
87            pw_log.printf("finalContainer size: %d\n", finalContainer.getSize());
88            pw_log.print("*---------------------------------------------------------------------------*\n\n");
89            Loader[] loadersCopy = loaders;
```

```
90            Improver[] improversCopy = improvers;
91            Resizer[] resizersCopy = resizers;
92            Cloner[] clonersCopy = cloners;
93
94            ////////////////// LOADERS ////////////////////////////////
95
96            int totalLoadedImages = 0;
97            for (Loader load : loadersCopy) {
98              totalLoadedImages += load.getLoadedImages();
99            }
100           pw_log.println("   Total loaders:\n");
101           pw_log.printf("     Loaded images: %d\n", totalLoadedImages);
102           pw_log.println("");
103           pw_log.printf("     Loaders: \n");
104           for (Loader loader : loadersCopy) {
105             pw_log.printf("        %s:\n", loader.getName());
106             pw_log.printf("          loaded images: %d\n", loader.getLoadedImages());
107           }
108           pw_log.println("");
109
110           ////////////////// IMPROVERS ////////////////////////////
111
112           int totalImprovedImages = 0;
113           for (Improver improver : improversCopy) {
114             totalImprovedImages += improver.getTotalImagesImprovedByThread();
115           }
116           pw_log.println("   Total Improvers:\n");
117           pw_log.printf("     Improved images: %d\n", totalImprovedImages);
118           pw_log.println("");
119           pw_log.printf("     Improvers: \n");
120           for (Improver improver : improversCopy) {
121             pw_log.printf("        %s:\n", improver.getName());
122             pw_log.printf("          improved images: %d\n",
    ↪ improver.getTotalImagesImprovedByThread());
123           }
124           pw_log.println("");
125
126           ////////////////// RESIZERS ////////////////////////////////
127
128           int totalResizedImages = 0;
129           for (Resizer resizer : resizersCopy) {
130             totalResizedImages += resizer.getTotalImagesResized();
131           }
132           pw_log.println("   Total Resizers:");
133           pw_log.println("");
134           pw_log.printf("     Resized images: %d\n", totalResizedImages);
135           pw_log.println("");
136           pw_log.printf("     Resizers: \n");
137           for (Resizer resizer : resizersCopy) {
138             pw_log.printf("        %s:\n", resizer.getName());
139             pw_log.printf("          resized images: %d\n", resizer.getTotalImagesResized());
```

```
140             pw_log.printf("          responsibility percentage in copied data over target data: %.2f
    ↪ %%\n",
141                     100 * (float) resizer.getTotalImagesResized() / totalLoadedImages);
142           }
143           pw_log.println("");
144
145           ////////////////////// CLONERS ////////////////////////////////
146
147           int totalClonedImages = 0;
148           pw_log.println("   Total Cloners:\n");
149           for (Cloner cloner : clonersCopy) {
150             totalClonedImages += cloner.getClonedImages();
151           }
152           pw_log.printf("     Cloned images: %d\n", totalClonedImages);
153           pw_log.println("");
154           pw_log.printf("     Cloners: \n");
155           for (Cloner cloner : clonersCopy) {
156             pw_log.printf("        %s:\n", cloner.getName());
157             pw_log.printf("          cloned images: %d\n", cloner.getClonedImages());
158             pw_log.printf("          responsibility percentage in taken data over target data: %.2f
    ↪ %%\n",
159                     100 * (float) cloner.getClonedImages() / totalLoadedImages);
160           }
161           pw_log.println("");
162
163           ////////////////////// Estados de los Hilos ////////////////////////////////
164
165           pw_log.println("   Threads State:\n");
166
167           for (Thread loaderThread : loadersThreads) {
168             pw_log.printf("      %s: %s\n", loaderThread.getName(),
    ↪ loaderThread.getState().name());
169           }
170
171           pw_log.println();
172
173           for (Thread improverThread : improversThreads) {
174             pw_log.printf("      %s: %s\n", improverThread.getName(),
    ↪ improverThread.getState().name());
175           }
176
177           pw_log.println();
178
179           for (Thread resizerThread : resizersThreads) {
180             pw_log.printf("      %s: %s\n", resizerThread.getName(),
    ↪ resizerThread.getState().name());
181           }
182
183           pw_log.println();
184
185           for (Thread clonerThread : clonersThreads) {
```

```
186             pw_log.printf("      %s: %s\n", clonerThread.getName(),
    ↪ clonerThread.getState().name());
187         }
188
189         pw_log.println();
190         pw_log.print("*--------------------------------------------------------------------------*\n\n");
191         pw_log.close();
192
193       } catch (IOException e) {
194         e.printStackTrace();
195       }
196     }
197   }
198
```

Código 10: Clase Resizer

```
1   package TP1;
2
3   import java.util.concurrent.TimeUnit;
4
5   // Clase encargada de reescalar las imágenes
6   public class Resizer implements Runnable {
7
8     private final InitContainer initContainer; // Contenedor inicial
9
10    private final String name; // Nombre del resizer
11
12    private Image lastImageResized; // Última imagen reescalada
13
14    private int totalImagesResized; // Cantidad de imágenes reescaladas por el hilo
15
16    private static int totalResizedImages = 0; // Cantidad de imágenes reescaladas por el proceso
17
18    private static boolean finishResized = false; // Flag que indica si el proceso de reescalado
    ↪ terminó
19
20    private final int targetAmountOfData; // Cantidad objetivo de imágenes
21
22    // Método que reescala las imágenes
23    public Resizer(InitContainer initContainer, String name , int targetAmountOfData) {
24      this.initContainer = initContainer;
25      this.name = name;
26      lastImageResized = null;
27      totalImagesResized = 0;
28      this.targetAmountOfData=targetAmountOfData;
29    }
30
31    // Getter de cantidad de imágenes reescaladas
32    public int getTotalResizedImages() {
33      return totalImagesResized;
```

```
34      }
35
36      // Se sobreescribe el run()
37      @Override
38      public void run() {
39        while (!finishResized) {
40          try {
41            lastImageResized = initContainer.getImage();
42            if (lastImageResized != null) {
43              if (lastImageResized.getAmIImproved()) {
44                if (lastImageResized.resize()) {
45                  System.out.println("Image: " + getlastImageResized().getId() + " resized by: "
46                  + Thread.currentThread().getName());
47                  increaseImageResizer();
48                  TimeUnit.MILLISECONDS.sleep(100);
49                  increaseTotalResizedImages();
50                  if(totalResizedImages == targetAmountOfData){
51                    finishResized=true;
52                  }
53                }
54              }
55            }
56          } catch (Exception e) {
57            e.printStackTrace();
58          }
59        }
60      }
61
62      // Getter del String name
63      public String getName() {
64        return name;
65      }
66
67      // Incrementador de imágenes reescaladas por el hilo
68      public void increaseImageResizer() {
69        totalImagesResized++;
70      }
71
72      // Incrementador de imágenes reescaladas por el proceso
73      public synchronized static void increaseTotalResizedImages(){
74        totalResizedImages++;
75      }
76
77      // Getter de la última imagen reescalada
78      public Image getlastImageResized() {
79        return lastImageResized;
80      }
81
82    }
83
```

Además de las clases mencionadas, se creó una excepción llamada **FullContainerException**, que si bien en la ejecución no se lanza, está como para que en caso de aparecer errores en el debug se pueda encontrar inmediatamente la causa.

Código 11: Excepción FullContainerException

```
1  package TP1;
2
3  /**
4  * Error a lanzar cuando se llena un contenedor
5  */
6  public class FullContainerException extends Exception {
7    /**
8    * @param errorMessage Mensaje de error a mostrar: tipicamente: "Contenedor lleno"
9    */
10   public FullContainerException (String errorMessage) {
11     super(errorMessage);
12   }
13 }
14
```

Todas las clases adjuntas resultan en un diagrama de clases UML como se ve a continuación:

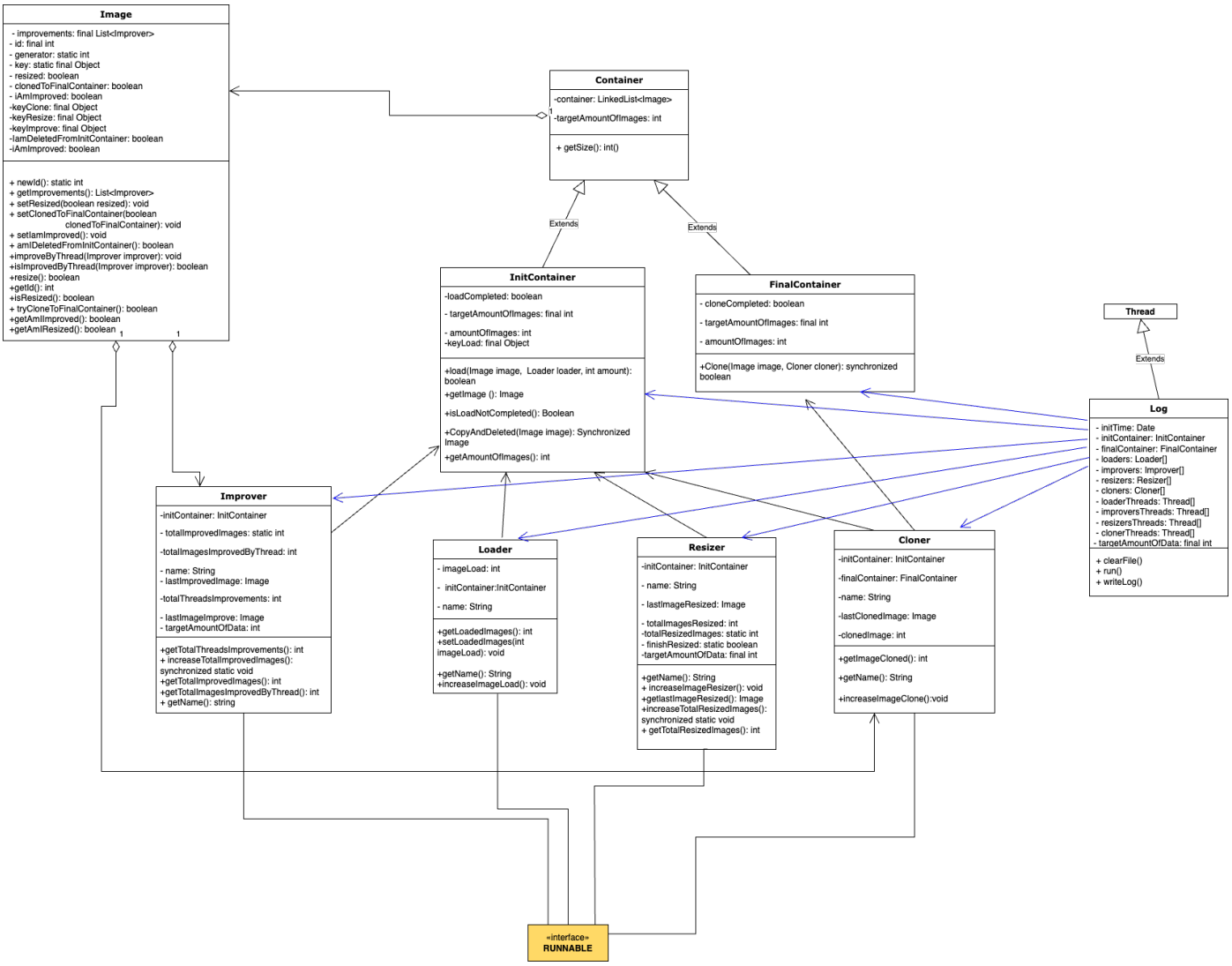

Figura 1: Diagrama de Clases

Y, el funcionamiento del código puede apreciarse en el siguiente diagrama de secuencia.

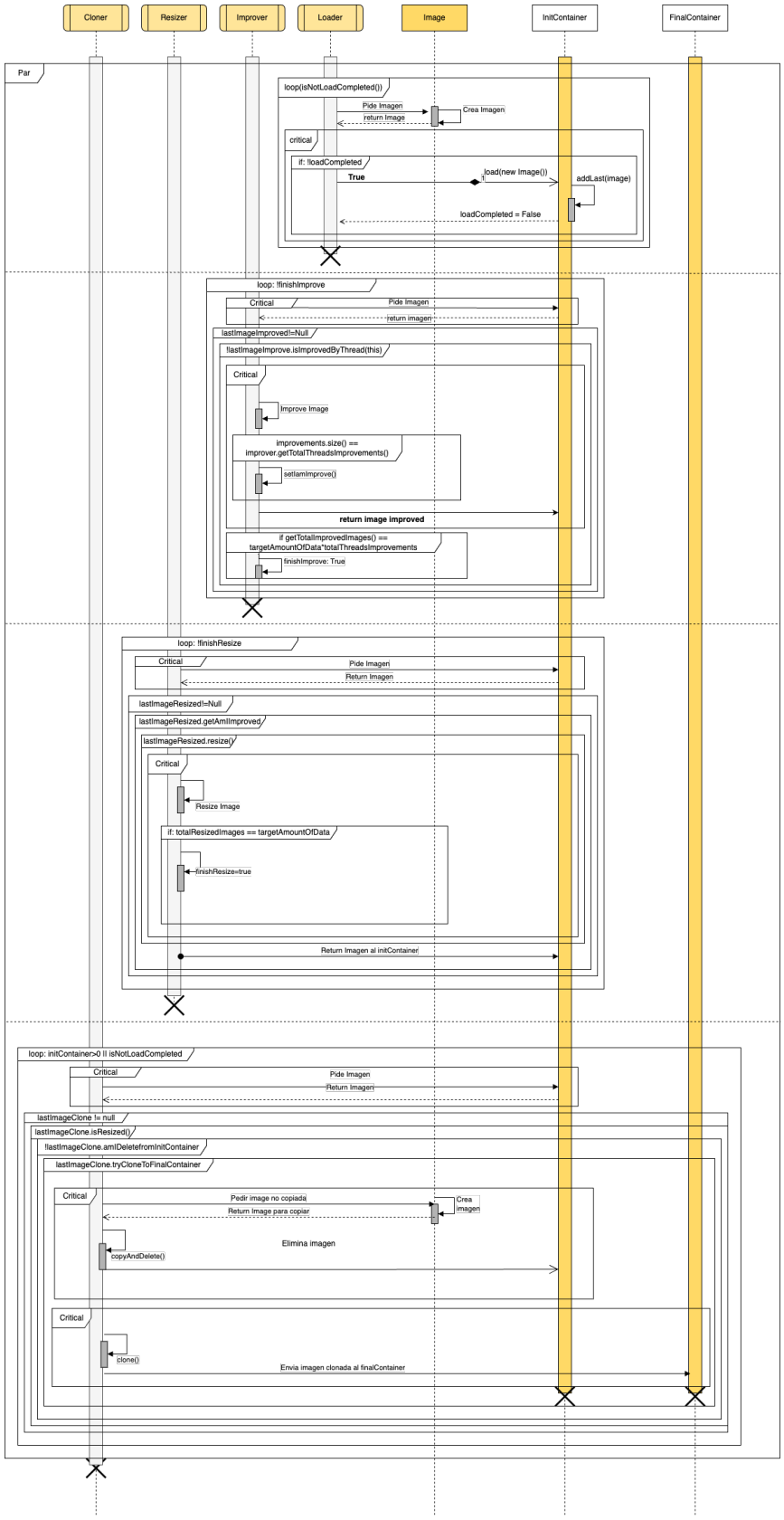

Figura 2: Diagrama de Secuencia

2.1.3. Estructura del código

A continuación se detallan los métodos principales de cada una de las clases implementadas:

- **Image**: Constructor de la clase. Recibe como parámetros: **List<Improver> improvements**, un **boolean resized**, **int id**, **boolean clonedToFinalContainer**, **boolean iAmImproved**.
  - **improveByThread(Improver improver)**: método que realiza la acción de mejorar la imagen por el Improver que recibe como parámetro.
  - **getAmIImproved()**: método que retorna true si la imagen ya fue mejorada por los tres improvers.
  - **amIDeletedfromInitContainer()**: método booleano que retorna **true** si la imagen fue eliminada del initContainer.
  - **isImprovedByThread(Improver improver)**: (bool), dice si un hilo del proceso Improver mejoró o no la imagen.
  - **resize()**: este método devuelve **true** si la imagen no fue reescalada por ningún hilo.
  - **isClonedToFinalContainer()**: (bool), este método retorna **true** si la imagen fue clonada en el contenedor final.
  - **tryCloneToFinalContainer()**: (bool), retorna **true** si la imagen no fue eliminada de initContainer.
  - **isResized()**: (bool), indica si la imagen fue reescalada o no.
- **Container**: esta clase representa el espacio de almacén de los datos.
  - **initContainer**: esta clase extiende de la clase **Container**. Aquí se cargan las imágenes iniciales.
    - **load(Image image)**: (void), carga una imagen en el contenedor.
    - **isLoadNotCompleted()**: (boolean), retorna **true** si la carga no está completa.
    - **copyAndDelete(Image image)**: (Image), copia la imagen y la borra del initContainer.
    - **getImage()**: método sincrónico que permite obtener una imagen aleatoria del contenedor.
  - **FinalContainer**: extiende de **Container**. Aquí se guardarán las imágenes ya mejoradas, reescaladas y copiadas.
    - **FinalContainer()**: constructor de la clase, que recibe un parámetro del tipo int con el número máximo de imágenes a trabajar.
    - **Clone(Image image, Cloner cloner)**: (synchronized boolean), este método consulta si el clonado está completo, siempre y cuando esté completo, la imagen no sea nula y la imagen haya sido recortada, agrega la imagen al FinalContainer. En caso de que la imagen sea la última, se detiene el trabajo de los Cloners.
- **Loader**: clase del proceso encargado de cargar las imágenes.
  - **Loader()**: constructor de la clase, recibe como parámetros un **InitContainer** y un String con el nombre de cada hilo.

  - **increaseImageLoad()**: aumenta en uno el contador de imágenes cargadas.

  - **Resizer**: clase del proceso que se encarga de reescalar las imágenes.
    - **Resizer()**: constructor de la clase, recibe un parámetro de tipo **InitContainer** y un String con el nombre.
    - **increaseImageResizer()**: (void), incrementa en uno la cantidad de imágenes reescaladas por la instancia (**totalImagesResized++**).
    - **increaseTotalResizedImages()**: (void), incrementa el contador de imágenes reescaladas por la clase.

- **Cloner**: clase del proceso encargado de clonar las imágenes del contenedor inicial.
  - **Cloner()**: constructor de la clase, recibe como parámetro un **InitContainer**, un **FinalContainer** y un string con el nombre del hilo.
  - **increaseImageClone()**: (void), método que incrementa en uno la cantidad de imágenes ya clonadas.

2.1.4. Herramientas utilizadas

Para escribir el código fuente del proyecto de utilizaron las siguientes herramientas:

- **IntelliJ IDEA Ultimate 2022.3**: para escribir el código y realizar las pruebas necesarias.
- **GitHub**: para mejorar la colaboración entre los participantes y poder mantener un control de versiones.
- **Diagrams.net**: para la realización de los diagramas UML requeridos.

3. Resultados obtenidos

3.1. Aspectos generales

Para la obtención de los tiempos de ejecución pedidos, se trabajó con diferentes tiempos de sleep dentro de las siguientes clases:

- Loader
- Improver
- Resizer
- Cloner

La salida del log **Estadisticas.txt** tiene el siguiente formato:

Código 12: Estadistica.txt

```
*---------------------------------------------------------------------------*
Execution time: 10.644 [Seg]
InitContainer size at this moment: 0
InitContainer size: 100
finalContainer size: 100
*---------------------------------------------------------------------------*

Total loaders:

   Loaded images: 100

   Loaders:
      TP1.Loader 0:
         loaded images: 50
      TP1.Loader 1:
         loaded images: 50

Total Improvers:

   Improved images: 300

   Improvers:
      TP1.Improver 0:
         improved images: 100
      TP1.Improver 1:
         improved images: 100
      TP1.Improver 2:
         improved images: 100

Total Resizers:

   Resized images: 100

```

```
        Resizers:
            TP1.Resizer 0:
                resized images: 35
                responsibility percentage in copied data over target data: 35.00 %
            TP1.Resizer 1:
                resized images: 35
                responsibility percentage in copied data over target data: 35.00 %
            TP1.Resizer 2:
                resized images: 30
                responsibility percentage in copied data over target data: 30.00 %

    Total Cloners:

        Cloned images: 100

        Cloners:
            TP1.Cloner 0:
                cloned images: 51
                responsibility percentage in taken data over target data: 51.00 %
            TP1.Cloner 1:
                cloned images: 49
                responsibility percentage in taken data over target data: 49.00 %

    Threads State:

        TP1.Loader 0 (Thread ID: 12): TERMINATED
        TP1.Loader 1 (Thread ID: 13): TERMINATED

        TP1.Improver 0 (Thread ID: 14): TERMINATED
        TP1.Improver 1 (Thread ID: 15): TERMINATED
        TP1.Improver 2 (Thread ID: 16): TERMINATED

        TP1.Resizer 0 (Thread ID: 17): TERMINATED
        TP1.Resizer 1 (Thread ID: 18): TERMINATED
        TP1.Resizer 2 (Thread ID: 19): TERMINATED

        TP1.Cloner 0 (Thread ID: 20): TERMINATED
        TP1.Cloner 1 (Thread ID: 21): TERMINATED

    *-------------------------------------------------------------------------*

```

3.2. Observaciones

Se definen los siguientes tiempos para las tareas:

- Loaders: 50ms
- Improvers: 100ms
- Resizers: 100 ms

- Cloner: 50 ms

El grupo decidió probar dos escenarios para la elección de los tiempos de trabajo de los procesos:

- Tareas totalmente secuencializadas (cota de tiempo superior).
- Tareas totalmente paralelizadas (cota de tiempo inferior).

Se procede con el análisis temporal.

3.2.1. Tarea absolutamente secuencial

Se tiene la siguiente suma:

$$50ms(Loaders) + 3 * 100ms(Improvers) + 100ms(Resizers) + 50ms(Cloners) = 500ms$$

Multiplicando lo anterior por las 100 imágenes a procesar dan como resultado $50000ms$.

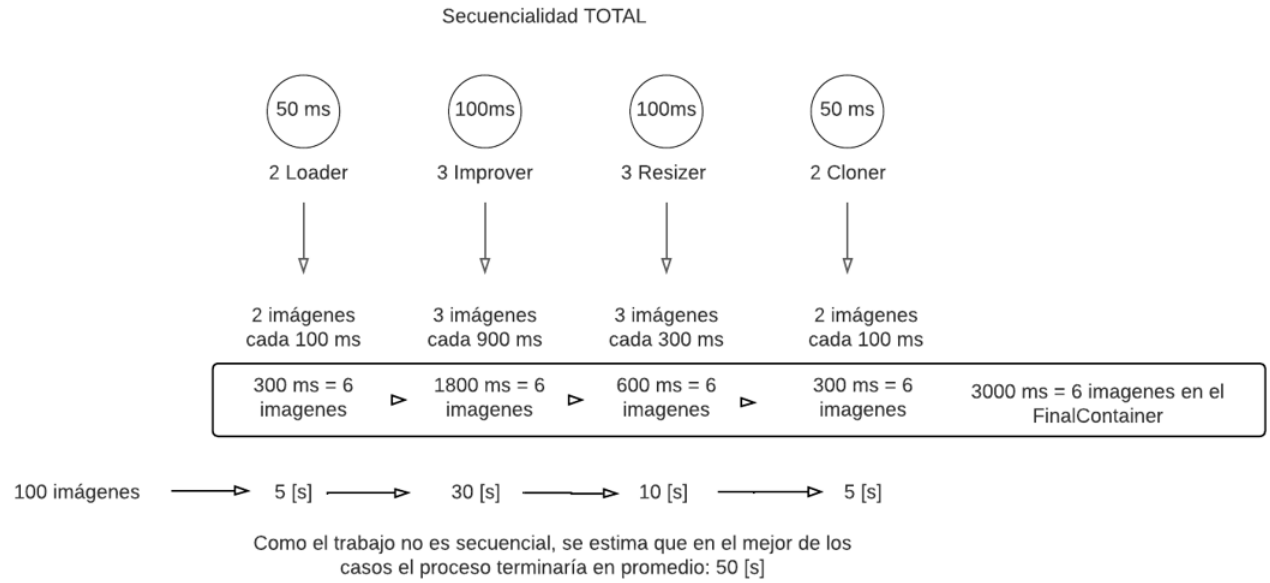

Figura 3: Caso de secuencialidad total

3.2.2. Tarea absolutamente paralela

Se plantea que el tiempo de procesamiento de cada imagen sería de $100ms$, por lo que para las 100 imágenes requeridas se tendría como tiempo de espera alrededor de 10 s. Todo esto teniendo en cuenta que el tiempo de sleep mencionado es el de los hilos Improvers. Ahora, colocando ese tiempo y paralelizando la ejecución de los resizers, las tareas deberían tardar ($\frac{100ms}{3}$), ya que se cuenta con 3 hilos resizers y se encargaron cada uno de un tercio de las imágenes a recortar (en el mejor de los casos).

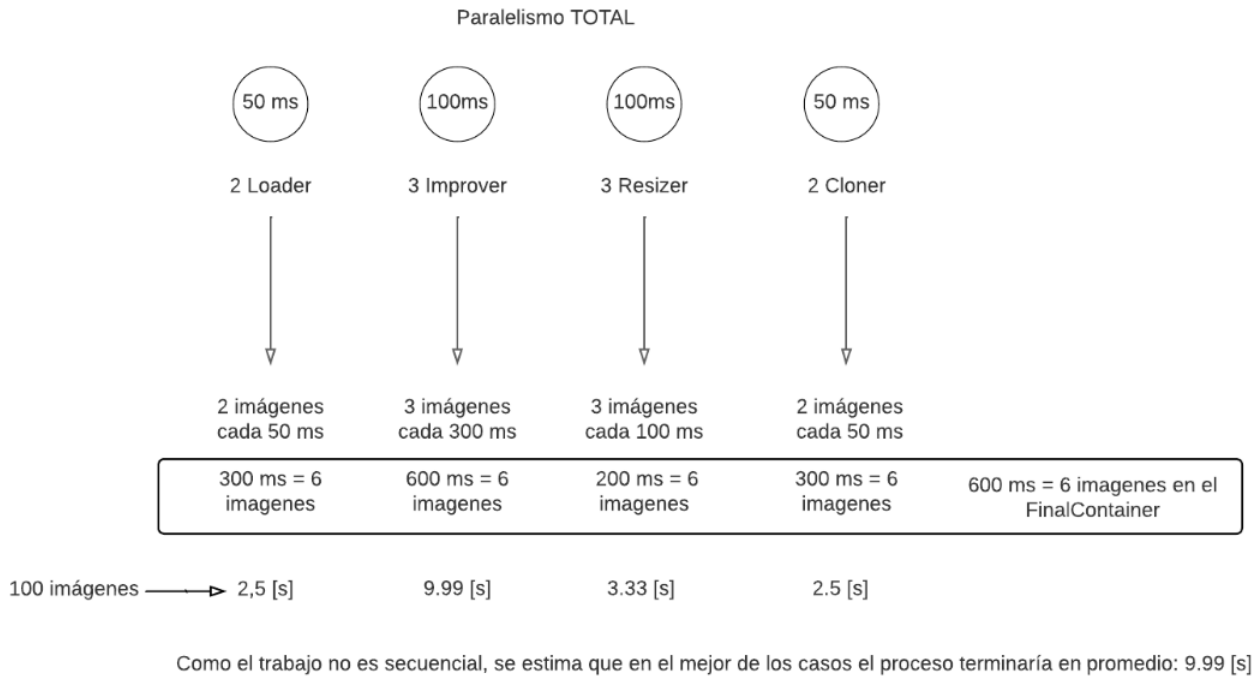

Figura 4: Caso de paralelismo total

Elección final

Luego de los análisis realizados sobre los casos extremos, se puede contar con una opción viable en términos de secuencialidad como la siguiente:

- Loaders: 10ms
- Improvers: 33ms
- Resizers: 20ms
- Cloners: 10ms

De esta forma es posible asegurarse que se todas las tareas están secuencializadas, el tiempo será menor a $200ms$ por cada imagen (20 s en total), pero estos tiempos afectaron en un sistema paralelo, ya que la ejecución completa duraría mucho menos de 10 segundos. Es por esto último que se considera un escenario como el siguiente:

- Loaders: 50ms
- Improvers: 100ms
- Resizers: 100ms
- Cloners: 50ms

De esta forma, al paralelizar los procesos la demora está fuertemente dominada por el tiempo de demora de los Improvers que mejoran 100 imágenes cada uno, dando un total de 10 segundos en el procesamiento de las 100 imágenes.

Cabe aclarar que los $100ms$ se colocan en el trabajo, son los tiempos para los Improvers, ya que si en un paralelismo total el tiempo más significativo por ejemplo se le asigna a los resizers se tardarían unos 3.33 s, ya que paralelizados cada uno de los resizers tomaría 33 imágenes y sumando así un total de 100. Esto último no sucede con los improvers ya que si o si tienen que mejorar 100 imágenes cada uno) tardando los $100ms$ de cada imagen, multiplicando por 100 imágenes, daría un tiempo de ejecución final de 10 s, cumpliendo con el objetivo de la consigna.

3.2.3. Comparación de tiempos de ejecución

Para comprobar cómo afectan los sleeps para cada hilo del programa, se realizó un análisis a partir de la ejecución del programa. Para ello, se fijó un tiempo de sleep para 3 de los 4 hilos que corren, este tipo fijado fue de $50ms$. Mientras que uno a uno se fue seleccionando un hilo y se lo corrió con diferentes tiempos de sleep:

1. $0ms$
2. $20ms$
3. $40ms$
4. $60ms$
5. $80ms$
6. $100ms$
7. $120ms$

Se obtuvieron así diferentes gráficos.

Primero, variando el tiempo de sleep del hilo Loader y manteniendo los otros contantes a $50ms$ se obtuvo:

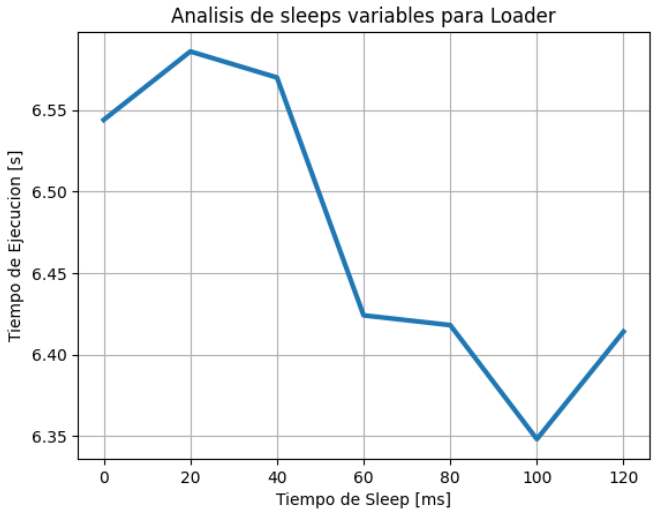

Figura 5: Variación de tiempos de sleep en Loader

Se aprecia en la figura anterior que esta modificación no afecta demasiado al tiempo de ejecución total del programa.

Ahora, modificando el Resizer:

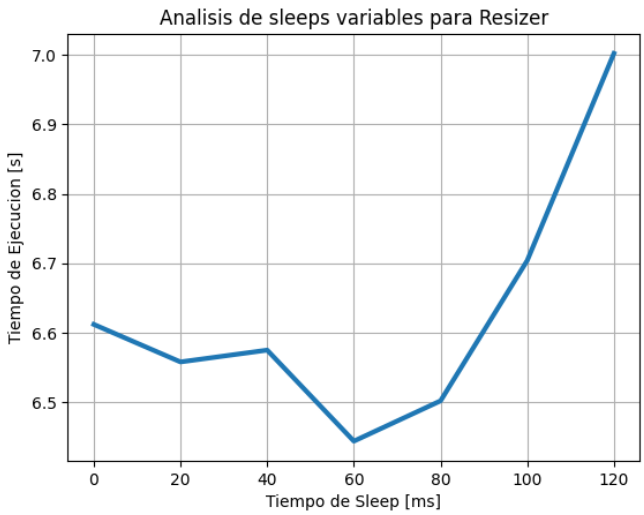

Figura 6: Variación de tiempos de sleep en Resizer

Nuevamente se aprecia cómo el tiempo de ejecución no varía tanto, aunque sí varía más que lo observado en el test anterior. A continuación se analiza la variación en Cloner.

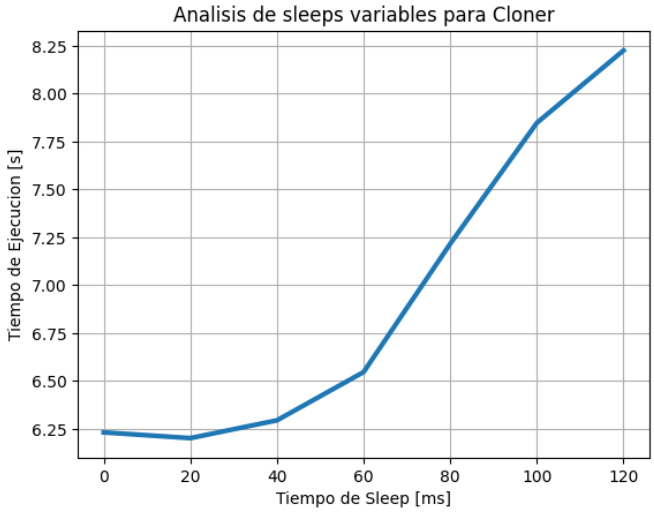

Figura 7: Variación de tiempos de sleep en Cloner

En este apartado, ya es posible apreciar un cambio superior a 1 *seg* entre el mínimo tiempo de sleep y el máximo. Finalmente, se comprueba el caso de tiempos de sleep variables para el Improver.

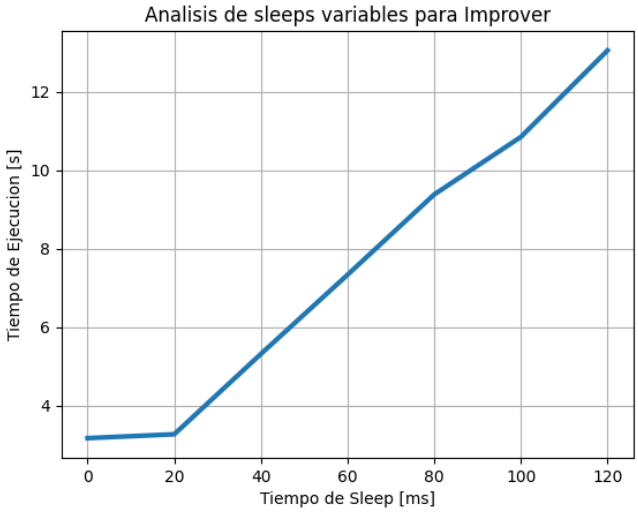

Figura 8: Variación de tiempos de sleep en Improver

En la figura anterior se aprecia un gran cambio en el tiempo de ejecución para la variación de tiempos de sleep en los hilos Improver. En la conclusión se explica qué está sucediendo. Para completar esta sección, se muestra un gráfico comparativo, en donde se aprecia claramente la variación observada en los tiempos de ejecución para la última prueba en comparación con las anteriores.

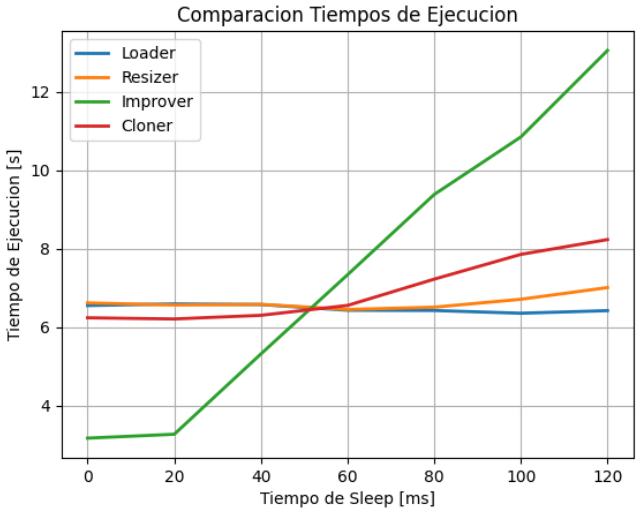

Figura 9: Comparación de la variación de tiempos

4. Conclusiones

Al final de la realización de este trabajo práctico, se pueden extraer las siguientes conclusiones:

- Se entendió la importancia de manejar de forma correcta la sincronización de los hilos, ya que, de lo contrario, si no son debidamente administradas las zonas en donde existe acceso a la memoria compartida (secciones críticas), se compromete la integridad de los datos y puede producir resultados inesperados.
- Se pudo determinar la mejor herramienta de manejo para la sincronización de los hilos, pudiendo así entender las diferencias existentes entre unas y otras. Para el caso del trabajo práctico, se utilizó **synchronized**. Se determinó que era la herramienta más adecuada para este contexto.
- Se encontró una relación aproximadamente lineal entre el tiempo de ejecución del programa y la cantidad de imágenes a procesar, de forma que es posible estimar el tiempo de ejecución para el programa. Se podría decir que se tiene una recta de ecuación

  $$t(n) = 0.1n$$

  En donde t representa el tiempo de ejecución, y n es la cantidad de imágenes a procesar.
- Durante el proceso de desarrollo fue posible observar la complejidad del manejo de las variables que intervienen en un proceso concurrente, y, que a medida que crece el programa se hace más dificil no cometer errores.
- Así mismo, fue posible comprobar que en el desarrollo de un programa, el trabajo en equipo en tiempo real agiliza ampliamente el el proceso de codificación.
- Se pudo ver que en un programa totalmente secuencial el tiempo de finalización del sistema crece mucho, pero tiene la ventaja de que la programación suele ser un poco más sencilla. Ahora, para el caso de la programación paralela, el tiempo de finalización es menor, pero la programación se complejiza bastante (ya que en el caso de los hilos se hace prácticamente imposible la independencia total entre los recursos que participan de las tareas de cada hilo). En el desarrollo de un proyecto (como bien podría ser el de este trabajo práctico), se debería evaluar la aplicación de una alternativa que esté a mitad de camino entre los extremos planteados, y esto es la programación concurrente, que se beneficia de los tiempos acotados gracias a momentos de paralelismo entre las tareas, y a la vez aprovecha momentos de secuencialidad para no generar problemas en el código, haciendo uso de secciones críticas.
- Luego de las pruebas realizadas sobre el tiempo de ejecución variando los tiempos de sleep de cada uno de los hilos, puedo verse que el único caso en que disminuye el tiempo de ejecución con el aumento de los tiempos de sleep es para el caso de Loaders. Se intuye que esto es porque el contenedor inicial tendrá menor cantidad de imágenes y por ende los hilos Improver tendrán menos trabajo que realizar, llegando más rápido a completar las 3 flags planteadas en el código. Esto hace que los Resizers puedan comenzar a trabajar antes y por consecuencia los Cloners también. Es decir, el aumento en la demora de Loaders implica que existen colas de menor tamaño de imágenes a procesar por cada uno de los hilos.

- Se aprecia que el tiempo de Improver es el más determinante en el tiempo de ejecución del programa, como puede verse claramente en la figura 8. Esto ocurre porque al tener cada imagen que ser mejorada por cada uno de los 3 hilos, se multiplica el delay que existe entre mejora y mejora, y, finalmente este delay afecta a la ejecución del programa completo.
- Un aumento en el tiempo de sleep para Cloner también implica que aumente el tiempo de ejecución, aunque en este caso, esa variación no es tan apreciable como en el caso anterior. Un aumento en el tiempo de sleep para Cloner implica un aumento en el tamaño de la cola de imágenes por clonar y borrar, siendo esta la causa principal del aumento de la demora en la ejecución.
- El cambio en los tiempos de ejecución del programa al variar el tiempo de sleep de los hilos Resizer es el que menos afecta a la ejecución total porque simplemente implica que los hilos Cloner tienen acceso a las imágenes apenas después. Además, como solo se reescala una vez cada imagen, esta demora no afecta tanto como una en un Improver, que debe ejecutarse 3 veces.
- Por último, con respecto a los tiempos de ejecución en diferentes ordenadores, se pudo apreciar que el tiempo de ejecución dados los mismos tiempos de sleep en cada hilo no varió demasiado a pesar de la diferencia en cantidad de nucleos y/o hilos de los procesadores. Esto se debe a que la mayor parte del tiempo los hilos están dormidos, por lo que la máquina virtual de Java no necesita estar constantemente reasignando recursos de forma de lograr una concurrencia muy marcada. Además, al realizar las primeras pruebas con sleeps muy chicos, la ejecución del programa era prácticamente instantánea, este resultado ayuda a comprobar la afirmación hecha.